JVM即时编译

通常，代码会先被Java虚机解释执行，之后反复执行的热点代码会被即时编译为机器码，直接运行在底层之上。

## Part1

### 分层编译模式

HotSpot虚机包含多个即时编译器，C1、C2和Graal。



Java7，我们需根据特性选择即时编译器。对于执行时间短的，或者对启动性能有要求的程序，我们采用便以效率较快的C1，对应参数-client

对于执行时间较长的，或者峰值想能有要求的，我们采用生成代码效率快的C2，对应参数-server



分层编译器五层：

注释：C1代码表示由C1编译器生成的机器码，C2代码表示C2编译器生成的机器码

1. 解释执行；
2. 执行不带profiling的C1代码；
3. 执行仅带方法调用次数以及循环回边执行次数profiling的C1代码；
4. 执行带所有profiling的C1代码；
5. 执行C2代码；

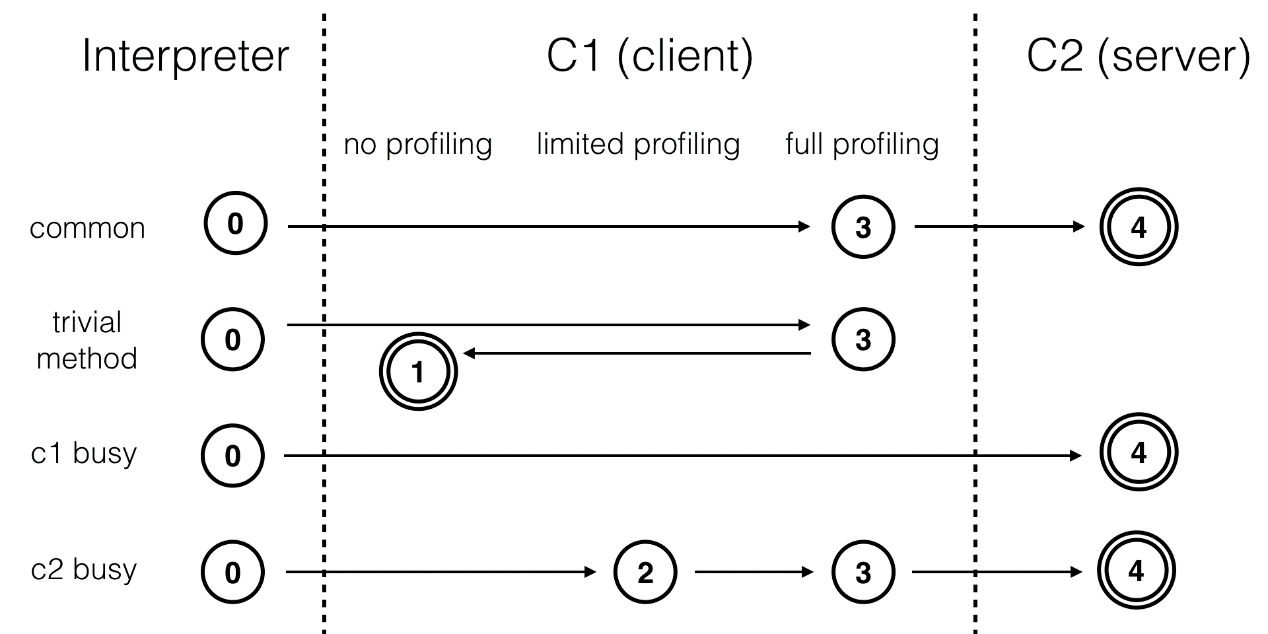
一般来说，C2代码的执行效率比C1代码的执行效率高30%以上。

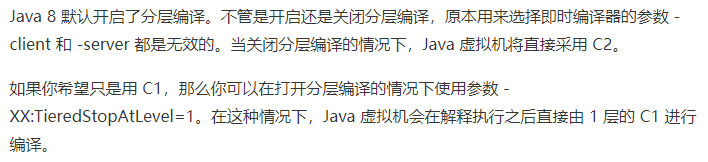
对于C1代码的三种状态，按照执行顺序分别试1>2>3。

1比2稍微高一些，2比3高30%，因为profiling越多，额外的性能开销越大。



在5个层次的执行状态中，1层和4层为终止状态。当一个方法被终止状态编译过后，如果编译后的代码并没有失效，那么jvm虚机并不会再次发出该方法的编译请求。





### 即时编译的触发

Java虚机根据方法的调用次数以及循环回边的执行次数来触发即时编译的。在0层、2层和3层执行状态进行profiling，其中就包含方法的调用次数和循环回边的执行次数。

循环回边：控制流图里面的概念。在字节码中，可以简单理解为往回跳转的指令。

**public static void** foo(**Object** o) {  
 **int** sum = 0;  
 **for** (**int** i = 0; i < 200; i++) {  
 sum += 1;  
 }  
}

public class cn.lxj.jvm40courses.course16\_comp.FooV1 {

public cn.lxj.jvm40courses.course16\_comp.FooV1();

Code:

0: aload\_0

1: invokespecial #1 // Method java/lang/Object."<init>":()V

4: return

public static void foo(java.lang.Object);

Code:

0: iconst\_0

1: istore\_1

2: iconst\_0

3: istore\_2

4: iload\_2

5: sipush 200

8: if\_icmpge 20

11: iinc 1, 1

14: iinc 2, 1

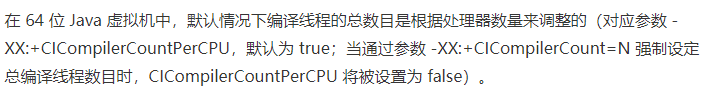
17: goto 4

20: return

}

字节码偏移量17回跳到了4字节码中。在解释执行中，每当运行一次该指令，java虚机就会将该方法的循环计数器加1；

实际上，Java虚机并不会对这些计数器进行同步操作，因此收集而来的执行次数并非精确值。无论如何，即时编译的触发并不需要非常精确的数值。只要改数值足够大，就能说明对应的方法包含热点代码。



Java虚机会将编译线程按照1：2的比例分配给C1和C2（至少各为1个）。对于一个四核机器来说，总编译线程为3，那么C1为1个线程，C2为2个线程。

对于四核及以上的机器，总的编译线程的数目为：

n = log2(N) \* log2(log2(N)) \* 3 / 2

其中 N 为 CPU 核心数目。

当启用分层编译时，及时编译具体的触发条件:

当方法调用次数大于由参数 -XX:TierXInvocationThreshold 指定的阈值乘以系数，或者当方法调用次数大于由参数 -XX:TierXMINInvocationThreshold 指定的阈值乘以系数，并且方法调用次数和循环回边次数之和大于由参数 -XX:TierXCompileThreshold 指定的阈值乘以系数时，便会触发 X 层即时编译。

触发条件为：

i > TierXInvocationThreshold \* s || (i > TierXMinInvocationThreshold \* s && i + b > TierXCompileThreshold \* s)

其中i为调用次数，b为循环回边次数。

### OSR编译（On-Stack-Replacement）

热点代码判断的因素：

1. 方法调用次数；
2. 循环回边执行次数；

为何维护两个计数器？

A:Java除了以方法为单位的及时编译器外，还有另一种以循环为单位的即时编译OSR。循环回边计数器便是用来触发这种编译。

**OSR**：在程序执行过程中，动态地替换掉Java方法栈帧，从而使得程序能够在非方法入口处，进行解释执行和编译后的代码之间的切换。去优化采用的技术也可称为OSR。

在不启用分层编译的情况下，触发OSR编译阈值是由参数-XX：CompileThreshold指定的阈值的倍数。

(OnStackReplacePercentage - InterpreterProfilePercentage)/100

其中 -XX:InterpreterProfilePercentage 的默认值为 33，当使用 C1 时 -XX:OnStackReplacePercentage 为 933，当使用 C2 时为 140。

也就是默认情况下，C1的OSR编译阈值为13500，而C2为10700。



通过以上学习，可以使用参数-XX：PrintCompilation在项目中打印即时编译情况：

88 15 3 CompilationTest::foo (16 bytes)

88 16 3 java.lang.Integer::valueOf (32 bytes)

88 17 4 CompilationTest::foo (16 bytes)

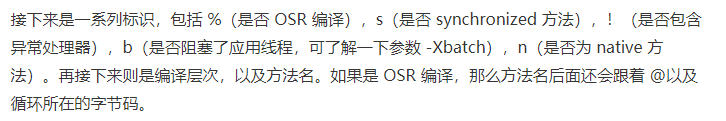
88 18 4 java.lang.Integer::valueOf (32 bytes)

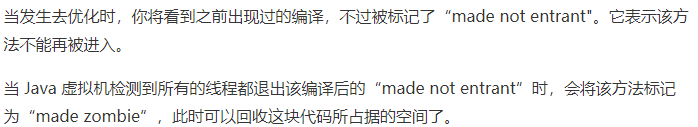
89 15 3 CompilationTest::foo (16 bytes) made not entrant

89 16 3 java.lang.Integer::valueOf (32 bytes) made not entrant

90 19 % 3 CompilationTest::main @ 5 (33 bytes)

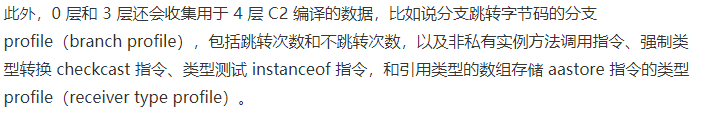
第一列表示时间，第二列表示java虚机维护的编译ID。





## Part2

### Profiling



因为收集Profiling信息消耗大量资源，并且性能下降30%。通常情况下，不会在解释执行的过程中，收集分支profile以及类型profile。只有在方法触发了C1编译后，java虚机才认为该方法有可能被C2编译，方才在该方法的C1代码中收集这些profile。

### 基于分支Profile的优化

分局跳转指令的分支profile，即时编译器可以将从未执行的分支剪掉，以避免编译这些很有可能用到的代码，从而节省编译时间，以及部署代码消耗的内存空间。此外，“剪枝”将精简程序的数据流，从而触发更多的优化。

### 基于类型Profile的优化

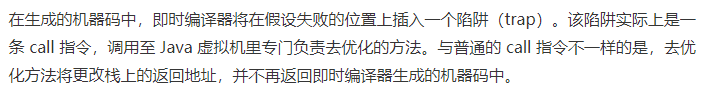
与分支Profile的优化一样，基于类型的Profile优化也同样是做出假设，从而精简控制流以及数据流。这两者核心都是假设。

对于分支Profile，即时编译器假设的是仅执行某一分支

对于类型Profile，即时编译器假设的是对象的动态类型仅为类型profile中的哪几个。

### 去优化

当假设失败，那么虚机给出的解决方案就是去优化，即从执行即时编译生成的机器码切回解释执行。



即时编译器会将java字节码转换成SSA IR。更确切的说，是一张包含控制流和数据流的IR图，每个字节码对应其中的若干个节点（注意：有些字节码并没有对应的IR节点）。然后，即时编译器在IR图上进行优化。

将每一种优化看做一个独立的图算法，它接收一个IR图，并输出经过转换后的IR图。整个编译器优化过程便是一个个优化串联起来的。

### See-of-nodes

HotSpot里的C2采用的是名为See-of-nodes的SSA IR。最大的特点是去除了变量的概念，直接采取变量所指向的值，来进行运算。

Graal的IR也是Sea-of-Nodes类型的，并且可以认为是C2 IR的精简版本。由于Graal的IR系统更加容易理解，工具相对来说比较安全、比较新，

可以使用工具<http://ssw.jku.at/General/Staff/TW/igv.html来展示具体的IR>图。

public static int foo(int count) {

int sum = 0;

for (int i = 0; i < count; i++) {

sum += i;

}

return sum;

}

